Manage transactions in the cloud, Part 1: Transaction basics and distributed transactions

See how two-phase commit distributed transactions work

Andre Fachat

In this first part of a two-part series, André Fachat discusses transaction basics and distributed transactions. Transactions can make error handling easy, not just with a single resource but also across resources. Here, André shows you how two-phase commit distributed transactions work and points out that even they are only "eventually consistent," if only by a small time interval. He also illustrates how a persistent transaction log in the transaction manager is a requirement for distributed transactions.

As a customer buying an item in a web shop, you want your order to be quickly processed and delivered. As a bank customer, you want to make sure your money does not mysteriously disappear during a transfer. In enterprise applications, classical transactions guarantee qualities such as consistency and isolation from other transactions. Distributed transactions provide such guarantees across multiple resources such as across your shop database and your enterprise resource planning (ERP) system. The transactional qualities are usually provided by your middleware infrastructure, and can make life easier for the average programmer.

In the cloud, however, middleware often does not guarantee such qualities; distributed transactions are usually not available, while at the same time the infrastructure can be much more volatile. Even small error rates can quickly scale up with your application and cause considerable support costs, so it is critical that you avoid or appropriately handle possible transaction management errors in scalable cloud applications.

In Part 1 of this two-part series, I provide some background on transaction processing, the qualities guaranteed by transactions, and I examine what is different in a cloud setup. In Part 2, I will show alternatives to classical transaction processing for calling non-transactional services.

"A main attribute of a transaction is its atomicity. In a real-world distributed transaction, this may not be fully achieved, and thus consistency can be broken as well."
General properties of a transaction

In the IT world, a transaction is generally considered to be a separate operation on information, which must succeed or fail as a complete unit. A transaction can never be partially complete. This is an important property. For example, financial operations rely on transactions in a way that the total amount of money does not change: If you take money from one account, you need to make sure the money is put into another account at the same time.

More formally, transactions are often associated with the four "ACID" properties:

- **Atomicity**— A transaction can only be "all or nothing." If it succeeds, all operations in a transaction are performed; if it fails, none of the operations is performed.
- **Consistency**— A transaction must bring the system from one consistent state to another consistent state. For example, any data written to a database must obey all of the constraints defined in the database (on commit).
- **Isolation**— Multiple concurrent transactions are isolated from each other in that they behave as if they were executed sequentially.
- **Durability**— Once a transaction is committed, the result of the transaction is there to stay, even in case of power losses or other errors.

While these are the theoretical principles of a transaction, in the real world, certain effects and optimizations can bend those principles for the sake of performance.

For example, while atomicity, consistency, and durability are strong goals in a transaction-processing system, isolation is often more relaxed due to the potential impact of strict isolation on performance. Different transaction isolation levels have been defined to determine which isolation failures are allowed in a system. These isolation levels appear within a single database, so we will not focus on that subject here.

A transaction is usually initiated by a transaction client calling a resource manager, which is often a database or a message queue. The resource manager manages resources such as database tables or rows in a database. The client opens a transaction, and, in possibly multiple calls, performs a number of operations on one or more resources in the system before it tries to commit the transaction (see Figure 1). When there is a single resource manager in a transaction, it can decide for itself whether a transaction is correct in terms of atomicity and consistency, and can, on commit, ensure durability.
**Figure 1. A typical transaction with client, resource manager, and resource**

![Diagram of a transaction with client, resource manager, and resource](image)

**Autocommit**

The discussion above uses explicit transaction demarcation with the `start()` and `commit()` operations, so multiple operations on one resource can be grouped into a single atomic change. During the transaction, the affected resources are locked against other changes. The decision whether to commit or roll back can then also depend on the outcome of other actions and possibly other resources.

When a transaction client needs to perform a single operation on a resource, there is no need to involve multiple calls to a resource manager. A single change on a single resource can be *autocommitted*—that is, it becomes effective immediately, as shown in Figure 2.
Figure 2. Explicit commit allows waiting for the outcome of other actions, but locks resources

The decision whether to autocommit or to use explicit commit or rollback for a single resource also plays a role in the error-handling patterns that are used for changes in multiple resources, as described below.

Long-lived versus short-lived transactions

The transaction properties described above are valid for transactions in the general sense. For example, even for a business transaction like a purchase with a credit card, you want those properties to hold. However, some transactions can take a long time. For example, when you check into a hotel, the hotel usually inquires electronically whether your account has enough money (that is, the limit on the card). But the charge to the account may not happen until days later when you check out. These are called *long-running* transactions — business transactions that are typically broken up into multiple technical transactions in order to avoid blocking (technical) resources in the transaction and resource managers.

Note that resource managers normally can’t lock resources across technical transactions. For a long-running business transaction to be able to lock a resource, the functional data model usually needs to be adapted to store this extra state of the resource.

For example, when a customer check into the hotel, a short-lived technical transaction reserves some amount on their account. The state of the reservation can be stored in a separate row in the database, and on reservation it is confirmed that the combined reservation doesn’t exceed the customer’s credit limit. On check-out, another short-lived technical transaction then exchanges the reservation with the actual money transfer. Each technical transaction may (within the limits of the isolation levels mentioned above) fulfill the ACID properties, but the business transaction does
not. The reserved money is not isolated from other transactions as it cannot be reserved by other business transactions.

Special techniques can be used to ensure atomicity and consistency in long-running business transactions. For example, reserving money for a credit card transaction is basically a form of pessimistic locking. We will discuss the use of such techniques in a cloud setting later in this article.

(For more discussion on this subject, see "Web-Services Transactions" at the SYS-CON Media website.)

**Local versus distributed transactions**

A local transaction involves only a single resource manager. But what happens when a transaction requires two or more resource managers? This can occur when you have to manage data in two databases, or send a message in a message queue depending on some state in the database. In such cases, just committing the first resource manager and then the second one will not work. If the second resource manager rolls back due to some constraint violations, how do you roll back the first resource manager if it is part of the transaction that is already committed?

The usual approach, and one that has been used very successfully, is to use a transaction manager, which coordinates the resource managers involved in a transaction. The transaction manager then uses a distributed transaction protocol, usually **two-phase commit**, to commit the transaction.

In two-phase commit, the transaction manager first asks all participating resource managers to guarantee that the transaction will succeed if committed. This is the **voting phase**. If all resource managers agree, then the transaction will be committed to all resource managers in the **commit phase**. If even one resource manager rejects the commit, all resource managers will be rolled back and consistency is guaranteed. Figure 3 shows a two-phase commit transaction.
Distributed transactions in the real world

A main attribute of a transaction is its atomicity. In a real-world distributed transaction, this may not be fully achieved, and thus consistency can be broken as well. Consider a two-phase commit transaction between a message queue and a database. A state is written to the database, and a related message is sent to the queue. Considering atomicity and consistency, one would expect that once a message is received from the queue, the database state will be consistent with it. This is the strong consistency model, which is guaranteed in classical transactions.

However, as the commit that is sent from the transaction manager to the two resource managers is sent one after the other, the change in the first resource might be seen before the change in the second resource. For example, a database as second resource may take more time to do the actual commit, and the message committed to the first resource manager could have already been received. Processing therefore will fail as the database state is not yet seen.

This does not happen often, but it does happen occasionally, and I have experienced it personally. (See "Two-phase commit race condition on WebLogic" and "Messaging/Database race conditions.") Therefore, even in classical distributed transactions, there is a window of inconsistency where transactions are only eventually consistent. See Figure 4.
In addition, the two-phase commit protocol requires that both phases be performed completely to release any blocked resources, such as database records. To ensure completeness of the transaction — for example, in the case of transaction manager outages or restarts — the transaction manager itself writes its own transaction log, which is kept when it is stopped. This log is then read when the transaction manager starts and pending transactions are completed. Therefore, a persistent transaction log, usually written to the file system, is required to support two-phase commit transactions in a transaction manager.

A two-phase commit transaction requires multiple remote calls just for the commit operation. Due to the perceived overhead of distributed transactions, these calls are often avoided, even in a classical JEE setup. However, they can be used when needed to ensure consistency across resource managers.

**Transactional Asynchronous (Messaging) Protocol**

One specific case where a distributed transaction is really well invested is when coupling message-oriented middleware with a database transaction. Making sure a message is taken out of a queue only when the database transaction succeeds is a very nice feature and reduces error-handling effort in the application. Making sure a message is sent only when the database transaction can be successfully committed also helps provide consistency between databases and messaging queues.

In such a setup, the distributed transaction is restricted to the "local" database and a messaging server, and does not include other resource managers. The messaging system then ensures the transactional properties between the sending and receiving application servers. This approach is
eventually consistent, as the receiving resource is changed only after the point where the change in the sending resource is committed.

**Figure 5. A messaging infrastructure can provide transactional eventual consistency guarantees**

Using message-oriented middleware, you can create an event-driven architecture with transactional qualities, and you can delegate much of the error handling to the transaction managers.

**Conclusion**

Here in Part 1, we have explored transaction basics and distributed transactions. Due to their properties, transactions can make error handling easy, not only with a single resource but especially across resources. I showed you how two-phase commit distributed transactions work and that even they are only "eventually consistent," if only by a small time interval. You also learned that a persistent transaction log in the transaction manager is a requirement for distributed transactions.

In many cloud deployments (CloudFoundry, for example), no persistent file system is available. A JEE application server runtime therefore cannot write its own transaction manager transaction log, because no two-phase commit transactions are available. Also, in some of the new runtime environments, such as a JavaScript server, runtimes do not even have a transaction manager available at all. (See "Microservices Best Practices for Java.")

**Part 2** will look at alternatives to distributed transactions that still provide some of the transactional guarantees. Using these alternatives, you can ensure transactional qualities even in a cloud application, and make sure the orders in your web shop are processed correctly.
Related topics

- Web-Services Transactions
- A Detailed Comparison of Enterprise JavaBeans (EJB) & The Microsoft Transaction Server (MTS) Models
- Two-phase commit race condition on WebLogic
- Messaging/Database race conditions
- Microservices Best Practices for Java

© Copyright IBM Corporation 2017
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)